Kauno technologijos universitetas

Informatikos fakultetas

Lygčių sistemų sprendimas

P170B115 Skaitiniai metodai ir algoritmai

Antras laboratorinis darbas

Projekto autorius
Gustas Klevinskas

Akademinė grupė
IFF-8/7

Vadovas
Andrius Kriščiūnas

Kaunas, 2020

Turinys

Įvadas

Užduoties variantas – 15.

1 užduotis. Tiesinių lygčių sistemų sprendimas

Duota tiesinė lygčių sistema

$$\begin{cases} x_2 + 2x_3 + x_4 = 2 \\ 6x_1 - 2x_2 + 3x_3 + 4x_4 = -15 \\ 3x_2 + 4x_3 - 3x_4 = 10 \\ -4x_2 + 3x_3 + x_4 = -2 \end{cases} \tag{1}$$

ir jos sprendimui nurodytas metodas: Gauso – Žordano.

2 užduotis. Netiesinių lygčių sistemų sprendimas

Duotos netiesinių lygčių sistemos

$$\begin{cases} e^{-\frac{\left((x_1+2)^2+2x_2^2\right)}{4}} - 0.1 = 0 \\ x_1^2 x_2^2 + x_1 - 8 = 0 \end{cases} \tag{2}$$

$$\begin{cases} 5x_1 + 4x_2 - 2x_3 + 22 = 0 \\ -5x_3^2 + 4x_1x_3 + 5 = 0 \\ -x_3^2 + x_4^3 + 2x_2x_4 + 1 = 0 \\ 3x_1 - 12x_2 + 3x_3 - 3x_4 - 63 = 0 \end{cases} \tag{3}$$

Sprendimo metodas: Niutono.

3 užduotis. Optimizavimas

Duotos n ($3 \le n$) taškų koordinatės ($-10 \le x \le 10,\ -10 \le y \le 10$).Srityje ($-10 \le x \le 10$, $-10 \le y \le 10$) reikia pridėti papildomų m ($3 \le m$) taškų taip, kad jų atstumai nuo visų kitų taškų (įskaitant ir papildomus) būtų kuo artimesni vidutiniam atstumui, o atstumas nuo koordinačių pradžios būtų kuo artimesnis nurodytai reikšmei S ($1 \le S$).

Pirma užduotis

Kadangi pirmos eilutės ir pirmo stulpelio elementas yra 0, aš prieš atlikdamas veiksmus sukeičiau pirmą eilutę su antra tiek A matricoje, tiek b vektoriuje.

Gautas sprendinys: [-2 1 1 -1].

Matricų išraiškos kiekviename žingsnyje:

1 lentelė. Matricų einamosios reikšmės

Žingsnis	A matrica	b vektorius
1	[[1 -0.33333333 0.5 0.66666667] [0 1 2 1] [0 3 4 -3] [0 -4 3 1]]	[-2.5 2 10 -2]
2	[[1 0 1.16666667 1] [0 1 2 1] [0 0 -2 -6] [0 0 11 5]]	[-1.83333333 2 4 6]
3	[[1 0 0 -2.5] [0 1 0 -5] [-0 -0 1 3] [0 0 0 -28]]	[0.5 6 -2 28]
4	[[1 0 0 0] [0 1 0 0] [-0 -0 1 0] [-0 -0 -0 1]]	[-2 1 1 -1]

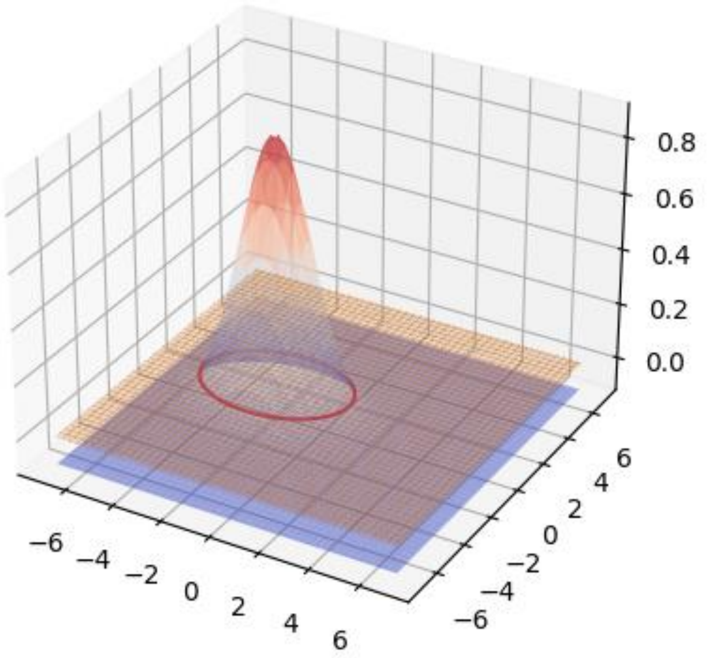
Įrašius gautus sprendinius į pradinę lygčių sistemą $[A][x] = [b]$ gauname b vektorių [-15 2 10 -2]. Jis sutampa su pradiniu vektoriumi.

Su numpy.linalg.solve() gautas rezultatas [-2 1 1 -1] sutampa su mano apskaičiuotu.

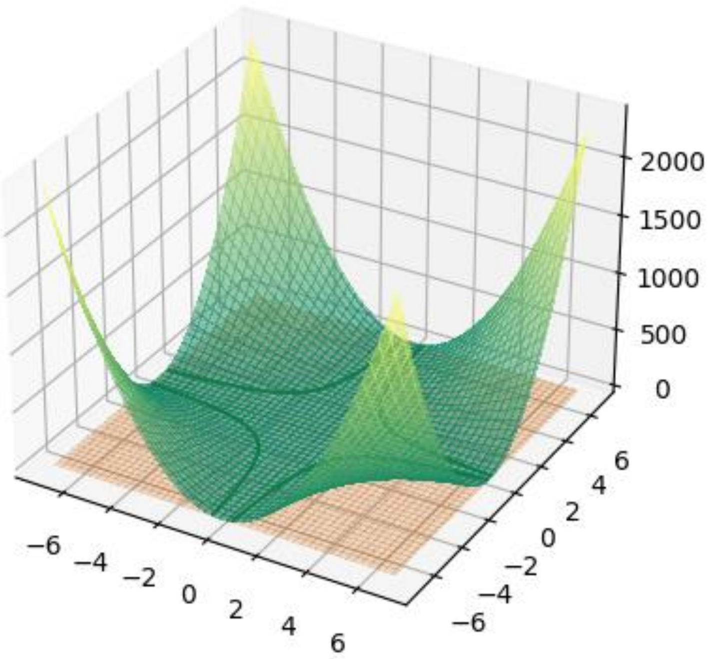
Antra užduotis

Pirma dalis

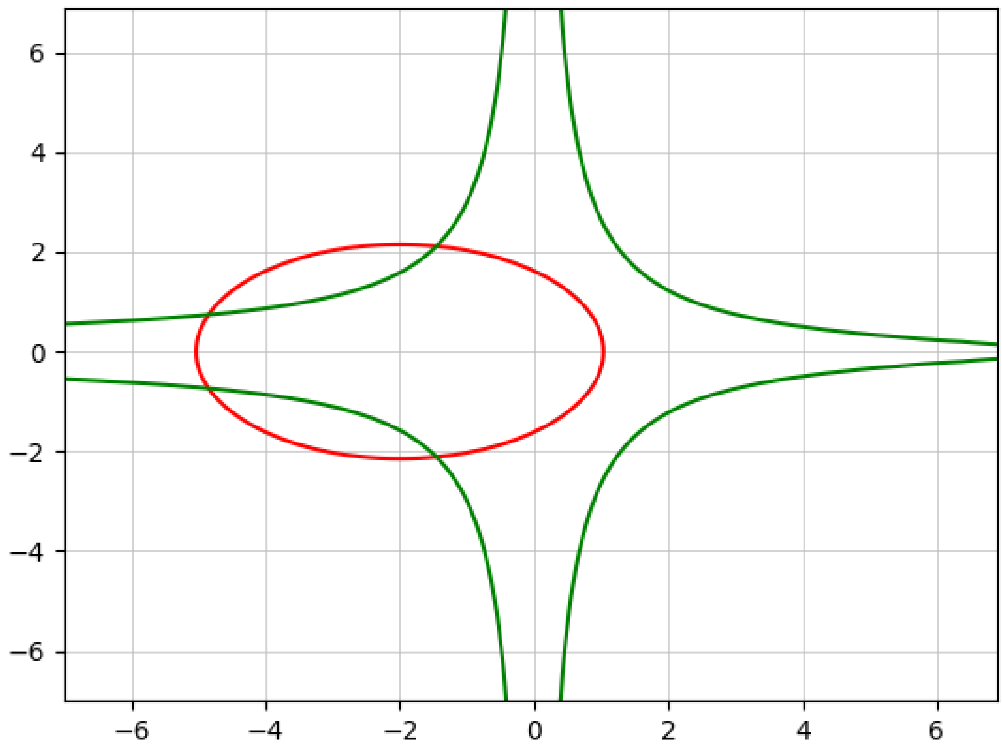
Paveikslėliuose 1 ir 2 pavaizduoti paviršiai $Z_1(x_1, x_2)$ ir $Z_2(x_1, x_2)$ atitinkamai.



1 pav. Z_1 paviršiaus grafikas



2 pav. Z_2 paviršiaus grafikas



3 pav. Paviršių susikirtimo taškai

Trečiame pav. pavaizduoti paviršių kontūrai ties Z = 0. Matome, kad šaknys apytiksliai yra: (-4.9, 1), (-4.9, -1), (-1.5, 2.1) ir (-1.5, -2.1).

Antroje lentelėje pavaizduoti keturi rezultatai gauti naudojant skirtingus pradinius artinius.

2 lentelė. Tikslesni lygčių sprendiniai

Pradinis artinys	Apskaičiuota šaknis	sympy.nsolve() šaknis	Iteracijų skaičius
[-1.5, 2.5]	[-1.45653423, 2.11127735]	[-1.45653418178908, 2.11127748962714]	4
[-2, -1]	[-1.45653418, -2.11127749]	[-1.45653418178908, -2.11127748962714]	5
[-5, -1]	[-4.84911305, -0.73922095]	[-4.84911304583501, -0.739220949388242]	4
[-6, 1]	[-1.45653418, 2.11127749]	[-1.45653418178908, 2.11127748962714]	6

Algoritmas tęsia skaičiavimus iki tol, kol visų funkcijų reikšmės einamajame taške neviršija 10^{-6}.

Antra dalis

Pradinis artinys: [1, 1, 1, 1].

Apskaičiuota šaknis: [0.60902393, -5.62485254, 1.27285475, 3.38128885].

Šaknis gauta su sympy.nsolve(): [0.609023930107426, -5.62485254073287, 1.27285474380282, 3.38128883684174].

Trečia užduotis

Pasirinktos reikšmės:

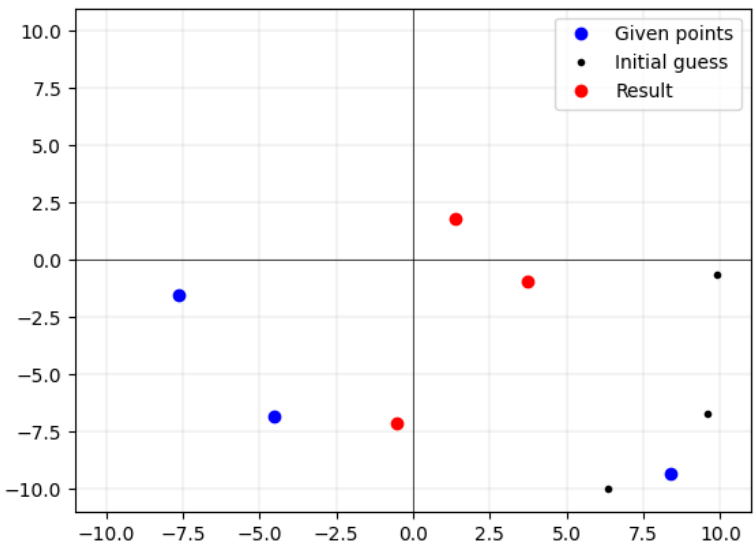
n (duotų taškų kiekis) = 3,

m (pridėtų taškų kiekis) = 3,

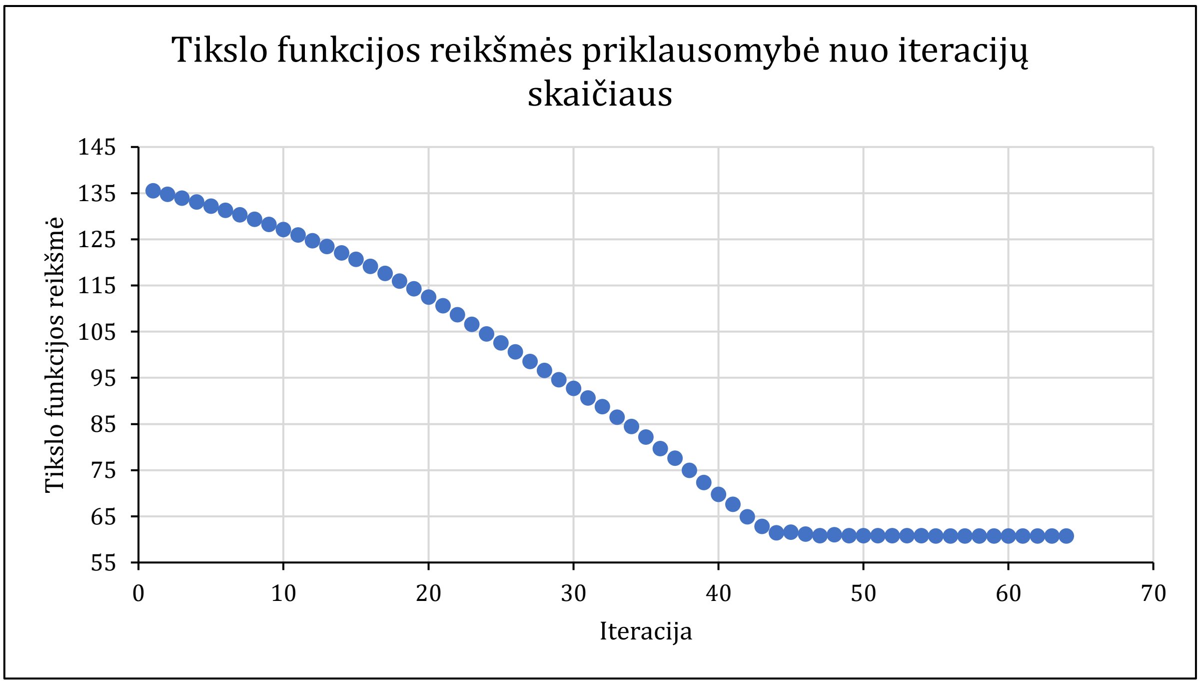
S = 1.

Programa parašyta taip, kad galima lengvai pakeisti šias reikšmes.

Šį optimizavimo uždavinį pasirinkau spręsti kvazi-gradientiniu metodu. Rezultatas gautas per 64 iteracijas.



4 pav. Optimizavimo uždavinio pradiniai artiniai ir rezultatai



5 pav. Tikslo funkcijos reikšmės priklausomybė nuo iteracijų skaičiaus

Mano aprašyta tikslo funkcija susideda iš trijų įverčių: atstumo tarp stacionarių ir pridėtų taškų, atstumo tarp pridėtų taškų tarpusavy ir pridėtų taškų atstumo nuo centro įskaičiuojant pasirinktą dydį S.

Algoritmas baigia darbą, kai žingsnis gradiento kryptimi tampa mažesnis nei 10^{-3}. Kiekvienoje iteracijoje apskaičiuojamas naujas artinys ir nauja tikslo funkcijos reikšmė. Jei ji mažesnė, žingsnį padidiname 5%, jei didesnė, kitaip tariant tikslas buvo peršoktas, grįžtame į praeitą artinį ir žingsnis sumažinamas 5 kartus.

Taikyto algoritmo parametrai:

- išvestinės žingsnis – 0.001;
- pradinis žingsnis gradiento kryptimi – 0.1.

Programinis kodas

L2_1.py

```
from sys import exit
import numpy as np


# Row 2 moved to row 1
A = [[6, -2, 3, 4],
     [0, 1, 2, 1],
     [0, 3, 4, -3],
     [0, -4, 3, 1]]
b = [-15, 2, 10, -2]

A_old = A
b_old = b

n = len(A)

for i in range(n):
    r1 = A[i][i]

    if abs(r1) < 1e-6:
        print('Leading element = 0, i = ', i)
        exit(1)

    for j in range(n):
        A[i][j] /= r1

    b[i] /= r1

    for j in range(n):
        if i != j:
            r2 = A[j][i]

            for k in range(i, n):
                A[j][k] -= r2 * A[i][k]

            b[j] -= r2 * b[i]


print("----- Calculated results -----")
print(b)

print("\n----- Check -----")
print("A * x =", np.matmul(A_old, b))
print("b =", b_old)

print("\n----- numpy solve() -----")
print(np.linalg.solve(A_old, b_old))
```

L2_2.py

```
import matplotlib.pyplot as plt
import numpy as np
import sympy
from matplotlib import cm


# ----- Solving -----
def newton(f, jacobian, xi):
    iterations = 0
    _xi = np.copy(xi)

    while max(np.abs(f(_xi))) > 1e-6:
        delta_x = np.linalg.solve(jacobian(_xi), np.negative(f(_xi)))
        _xi = np.add(_xi, delta_x)
        iterations += 1

    return _xi, iterations


# ----- Part 1 -----
x1, x2 = sympy.symbols('x1 x2')

fun1 = sympy.exp(-((x1 + 2) ** 2 + 2 * x2 ** 2) / 4) - 0.1
fun2 = x1 ** 2 * x2 ** 2 + x1 - 8

f1_diff_x1 = sympy.lambdify([x1, x2], fun1.diff(x1))
f1_diff_x2 = sympy.lambdify([x1, x2], fun1.diff(x2))
f2_diff_x1 = sympy.lambdify([x1, x2], fun2.diff(x1))
f2_diff_x2 = sympy.lambdify([x1, x2], fun2.diff(x2))

f1 = sympy.lambdify([x1, x2], fun1)
f2 = sympy.lambdify([x1, x2], fun2)


def f_arr(xi):
    return [
        f1(xi[0], xi[1]),
        f2(xi[0], xi[1])
    ]


def jacobian(xi):
    substitutions = [(x1, xi[0]), (x2, xi[1])]
    j = sympy.Matrix([[fun1, fun2]]).jacobian([x1, x2]).subs(substitutions)
    return np.array(j).astype(np.float64)


xi = [-6.0, 1.0]
newton_root1, iterations = newton(f_arr, jacobian, xi)
nsolve_root = sympy.nsolve((fun1, fun2), (x1, x2), xi)
print("----- Part 1 -----")
print("Iterations: ", iterations)
print("Root:       ", newton_root1)
print("nsolve root: [{}, {}]".format(nsolve_root[0], nsolve_root[1]))


# ----- Part 2 -----
x1, x2, x3, x4 = sympy.symbols('x1 x2 x3 x4')

fun1 = 5 * x1 + 4 * x2 - 2 * x3 + 22
fun2 = -5 * x3 ** 2 + 4 * x1 * x3 + 5
fun3 = -x3 ** 2 + x4 ** 3 + 2 * x2 * x4 + 1
fun4 = 3 * x1 - 12 * x2 + 3 * x3 - 3 * x4 - 63

f1 = sympy.lambdify([x1, x2, x3, x4], fun1)
f2 = sympy.lambdify([x1, x2, x3, x4], fun2)
f3 = sympy.lambdify([x1, x2, x3, x4], fun3)
f4 = sympy.lambdify([x1, x2, x3, x4], fun4)


def f_arr(xi):
    return [
```

```
        f1(xi[0], xi[1], xi[2], xi[3]),
        f2(xi[0], xi[1], xi[2], xi[3]),
        f3(xi[0], xi[1], xi[2], xi[3]),
        f4(xi[0], xi[1], xi[2], xi[3]),
    ]


def jacobian(xi):
    substitutions = [(x1, xi[0]), (x2, xi[1]), (x3, xi[2]), (x4, xi[3])]
    j = sympy.Matrix([[fun1, fun2, fun3, fun4]]).jacobian([x1, x2, x3, x4]).subs(substitutions)
    return np.array(j).astype(np.float64)


xi = [1.0, 1.0, 1.0, 1.0]
newton_root2, iterations = newton(f_arr, jacobian, xi)
nsolve_root = sympy.nsolve((fun1, fun2, fun3, fun4), (x1, x2, x3, x4), xi)
print("----- Part 2 -----")
print("Iterations: ", iterations)
print("Root:       ", newton_root2)
print("nsolve root: [{}, {}, {}, {}]".format(nsolve_root[0], nsolve_root[1], nsolve_root[2],
nsolve_root[3]))


# ----- Plotting -----
X1 = np.arange(-7, 7, 0.1)
X2 = np.arange(-7, 7, 0.1)
XX1, XX2 = np.meshgrid(X1, X2)

Z1 = np.exp(-((XX1 + 2) ** 2 + 2 * XX2 ** 2) / 4) - 0.1
Z2 = XX1 ** 2 * XX2 ** 2 + XX1 - 8

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(XX1, XX2, Z1, cmap=cm.coolwarm, alpha=0.5)
ax.plot_surface(XX1, XX2, np.zeros(np.shape(Z1)), antialiased=False, alpha=0.2)
ax.contour(X1, X2, Z1, levels=0, colors='red')

fig = plt.figure()
ax = fig.gca(projection='3d')
ax.plot_surface(XX1, XX2, Z2, cmap=cm.summer, antialiased=False, alpha=0.5)
ax.plot_surface(XX1, XX2, np.zeros(np.shape(Z1)), antialiased=False, alpha=0.2)
ax.contour(X1, X2, Z2, levels=0, colors='green')

fig = plt.figure()
ax = fig.gca()
ax.grid(color='#C0C0C0', linestyle='-', linewidth=0.5)
ax.contour(X1, X2, Z1, levels=0, colors='red')
ax.contour(X1, X2, Z2, levels=0, colors='green')
plt.plot(newton_root1[0], newton_root1[1], 'o')

plt.show()
```

L2_3.py

```
import numpy as np
import random
from sys import exit
import matplotlib.pyplot as plt


N = 3
M = 3
S = 1


def generate_points(amount):
    points = []

    for i in range(amount):
        points.append([random.uniform(-10, 10), random.uniform(-10, 10)])

    return np.array(points)


# Average distance
def avg_dist(stat_arr, new_arr):
    total = 0
    amount = 0

    for point in stat_arr + new_arr:
        total += np.sqrt(point[0] ** 2 + point[1] ** 2)
        amount += 1

    return total / amount


# Distance between stationary and new points
def v1(stat_arr, new_arr, d):
    total = 0

    for new_point in new_arr:
        for stat_point in stat_arr:
            total += abs(np.sqrt((new_point[0] - stat_point[0]) ** 2 + (new_point[1] - stat_point[1])
** 2) - d)

    return total


# Distance between new points
def v2(new_arr, d):
    total = 0

    for i in range(M):
        for j in range(i + 1):
            total += abs(np.sqrt((new_arr[i][0] - new_arr[j][0]) ** 2 + (new_arr[i][1] -
new_arr[j][1]) ** 2) - d)

    return total


# Distance from center
def v3(new_arr):
    total = 0

    for point in new_arr:
        total += abs(np.sqrt(point[0] ** 2 + point[1] ** 2) - S)

    return total


def psi(stat_arr, new_arr):
    d = avg_dist(stat_arr, new_arr)

    return v1(stat_arr, new_arr, d) + v2(new_arr, d) + v3(new_arr)
```

```python
def psi_prime(stat_arr, new_arr, i, j):
    h = 0.001
    increased_arr = np.copy(new_arr)
    increased_arr[i][j] += h

    psi1 = psi(stat_arr, new_arr)
    psi2 = psi(stat_arr, increased_arr)

    if psi2 - psi1 == 0:
        print("Derivative is 0")
        exit(1)

    return (psi2 - psi1) / h


def gradient(stat_arr, new_arr):
    g = []

    for i in range(M):
        g.append([
            psi_prime(stat_arr, new_arr, i, 0),
            psi_prime(stat_arr, new_arr, i, 1)
        ])

    return np.array(g) / np.linalg.norm(g)


def is_beyond_10(points):
    for i in points:
        if abs(i[0]) > 10 or abs(i[1]) > 10:
            return True

    return False


def solve():
    stat_points = generate_points(N)
    new_points = generate_points(M)

    initial_points = new_points
    step = 0.1
    iteration_count = 0
    old_psi = psi(stat_points, new_points)

    while step > 1e-3:
        iteration_count += 1

        old_points = new_points
        new_points = new_points - step * gradient(stat_points, new_points)

        if is_beyond_10(new_points):
            print("New points beyond 10")
            print(iteration_count, "iterations")
            return stat_points, initial_points, old_points

        new_psi = psi(stat_points, new_points)

        if new_psi > old_psi:
            step /= 5
            new_points = old_points
        else:
            step += step * 0.05

        old_psi = new_psi

    print(iteration_count, "iterations")
    return stat_points, initial_points, new_points


given_points, initial_guess, result = solve()

print("Initial guess")
print(initial_guess)
print("Result")
```

```
print(result)

plt.xlim(-11, 11)
plt.ylim(-11, 11)
plt.axvline(linewidth=0.5, color='k')
plt.axhline(linewidth=0.5, color='k')
plt.grid(linewidth=0.2)

given_point = None
initial_point = None
result_point = None

for point in given_points:
    given_point, = plt.plot(point[0], point[1], 'bo')

for point in initial_guess:
    initial_point, = plt.plot(point[0], point[1], 'ko', markersize=3)

for point in result:
    result_point, = plt.plot(point[0], point[1], 'ro')


plt.legend(
    [given_point, initial_point, result_point],
    ["Given points", "Initial guess", "Result"]
)

plt.show()
```